Optimization of Time-Dependent Particle Tracing Using Tetrahedral Decomposition

David N. Kenwright and David A. Lane
Computer Sciences Corporation
NASA Ames Research Center, Mail Stop T27A-2, Moffett Field, CA 94035

Abstract

An efficient algorithm is presented for computing particle paths, streak lines and time lines in time-dependent flows with moving curvilinear grids. The integration, velocity interpolation, and step size control are all performed in physical space which avoids the need to transform the velocity field into computational space. This leads to higher accuracy because there are no Jacobian matrix approximations, and expensive matrix inversions are eliminated. Integration accuracy is maintained using an adaptive step size control scheme which is regulated by the path line curvature. The problem of point location and interpolation in physical space is simplified by decomposing hexahedral cells into tetrahedral cells. This enables the point location to be done analytically and substantially faster than with a Newton-Raphson iterative method. Results presented show this algorithm is up to six times faster than particle tracers which operate on hexahedral cells and produces almost identical traces.

1. Introduction

Unsteady particle tracing is a relatively new visualization technique that has emerged because of the need to visualize unsteady or time-dependent datasets. Whereas steady-state flow simulations only require one set of grid and solution data to describe a flow, unsteady flow simulations may comprise of hundreds or thousands of time steps of data; each having an associated grid and solution file. The size of these datasets can run into hundreds of gigabytes [1,2].

The techniques used for visualizing unsteady flows in experimental fluid mechanics include path lines, time lines and streak lines. They are generated by injecting foreign material such as particles, dye or smoke into the fluid:

- *path line*: generated by tracing the path of a single particle (also called a particle path).
- *time line*: generated by tracing a line of particles which are all released at the same time.
- *streak line*: generated by continuously injecting particles from a fixed location.

In this study we were interested in generating streak lines from large time-dependent computational fluid dynamics (CFD) simulations. In computational flow visualization, streak lines are generated by releasing particles at discrete intervals, usually in accordance with the simulation time steps. This continuous injection leads to a rapid growth of the number of particles in the flow, all of which must be tracked until they leave the flow-field or until the simulation ends, which ever comes first. It is not uncommon for there to be several thousand active particles in an unsteady flow, so it is essential that the advection or tracing process is as efficient as possible.

A time-accurate particle tracing algorithm is presented in this paper which offers improved scalar performance without sacrificing accuracy. The algorithm can be used to compute path lines, time lines, or streak lines in structured or unstructured grids.

2. Previous work

Many algorithms have been presented for particle tracing in steady flows yet relatively few consider the extension to unsteady flows. This extension is not always trivial because the time varying nature of the flow and grid adds complexity to almost every part of the algorithm [2]. Also, techniques used to speed-up steady flow algorithms, such as loading the entire solution into memory or pre-evaluating computational space velocities, are not possible with large time-dependent datasets.

To improve particle tracing performance, Shirayama [3] transformed the velocity vector field into a uniform (computational) space and decomposed hexahedral cells into tetrahedra. The particles were then traced in computational space. Shirayama's approach significantly reduces the number of calculations although accuracy suffers because of approximations made in the transformation to computational space. This problem is discussed further in section 3.

Significant gains in performance can be achieved by particle tracing on parallel [4,5] and multi-processor systems [6]. Particle tracing is easily parallelized since each particle can be traced independently. The algorithm presented here improves scalar performance although could be parallelized to achieve even greater speed-ups.

(See color plates, page CP-38)

1070-2385/95 $4.00 © 1995 IEEE

3. Physical vs computational space

Particle tracing algorithms for structured curvilinear grids can divided into those that trace in computational space and those that trace in physical space. Computational space schemes require the curvilinear grid and the associated velocity field to be mapped onto a uniform rectangular grid using Jacobian transformations. This can either be done as a preprocessing step for the whole grid [7] or locally on-the-fly as required by the tracking process [8]. Particles are then advected in the logical grid space rather than in physical space.

The main disadvantage of tracing in computational space is that the transforming Jacobian matrices are only approximations and so the transformed vector field may be discontinuous. Also, if there are irregularities in the grid, such as cells with collapsed edges, the transformed velocities may be infinite [9]. Analyses by Sadarjoen et al. [10] and Hultquist [11] in steady flows have shown that this mapping technique produces significant errors in distorted curvilinear grids. Calculating Jacobian matrices in unsteady flows with moving grids is more inaccurate because each matrix has three additional time-dependent terms which must be approximated [12].

Physical space tracing schemes are more accurate because the interpolation and integration processes are done in Cartesian space which eliminates the need to evaluate Jacobian matrices and transform the velocity field. The only disadvantage is that the task of locating particles is more complicated and hence more expensive. This problem is addressed in this paper. An explicit point location technique is presented which has yielded a significant speed-up over the most common point location technique; the Newton-Raphson iterative method.

4. Physical space tracing algorithm

Physical space tracing algorithms proceed by firstly searching out the element or cell which bounds a given point. This is termed the *cell search* or *point location* process. Once found, the velocity is evaluated at that point by interpolating the nodal velocities. In unsteady flows, the velocity components usually need to be interpolated temporally as well as spatially. This necessitates loading two or more time steps of data into memory. Intermediate positions of the grid may also have to be interpolated if the grid changes in time. The particle's path is determined by solving the differential equation for a field line:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}(\,\mathbf{r}(t),t\,) \qquad (1)$$

where $\mathbf{r}$ is the particle's location and $\mathbf{v}$ the particle's velocity at time t. Integrating (1) yields:

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \int_{t}^{t+\Delta t} \mathbf{v}(\,\mathbf{r}(t),t\,)\,dt \qquad (2)$$

The integral term on the right hand side of this equation can be evaluated numerically using a multi-stage method (e.g., Runge-Kutta, Bulirsch-Stoer) or a multi-step method (e.g., Backwards differentiation, Adams-Bashforth). Issues concerning the accuracy and stability of these methods are discussed by Darmofal and Haimes [13]. Regardless of how it is solved, the end result is a displacement which when added to the current position, $\mathbf{r}(t)$, gives the new particle location at time $t+\Delta t$.

The essential steps in a time-dependent particle tracing algorithm are as follows:

1. Specify the seed point for a particle in physical space, (x,y,z,t).
2. Perform a point location to locate the cell that contains the point.
3. Evaluate the cell's velocities and coordinates at time t by interpolating between simulation time steps.
4. Interpolate the velocity field to determine the velocity vector at the current position, (x,y,z).
5. Integrate the local velocity field using equation (2) to determine the particle's new location at time $t+\Delta t$.
6. Estimate the integration error. Reduce the step size and repeat the integration if the error is too large.
7. Repeat from step 2 until particle leaves flow field or until t exceeds the last simulation time step.

5. Point location in tetrahedral cells

The core problem in all particle tracers is: given an arbitrary point $\mathbf{x}$ in physical space, which cell does this point lie in and what are its natural coordinates. The natural coordinates are local non-dimensional coordinates (see Figure 1) and are different from the computational coordinates which are globally defined.

The widely used trilinear interpolation function (3) provides the opposite mapping to that required, that is, it determines the location of $\mathbf{x}$ from a given natural coordinate (ξ,η,ζ). Unfortunately, equation (3) cannot be inverted analytically because of the non-linear products, so it has to be solved numerically using an iterative scheme such as the Newton-Raphson method.

$$\begin{aligned}\mathbf{x}(\xi,\eta,\zeta) = {} & [\,(\mathbf{x}_{000}(1-\xi) + \mathbf{x}_{100}\xi)(1-\eta) + \\ & (\mathbf{x}_{010}(1-\xi) + \mathbf{x}_{110}\xi)\eta\,](1-\zeta) + \\ & [\,(\mathbf{x}_{001}(1-\xi) + \mathbf{x}_{101}\xi\,)(1-\eta) + \\ & (\mathbf{x}_{011}(1-\xi) + \mathbf{x}_{111}\xi)\eta\,]\zeta \end{aligned} \qquad (3)$$

An alternative point location scheme has been developed based on tetrahedral elements which allows the natural coordinates to be evaluated directly from the physical coordinates. The mathematical basis of this method will now be described.

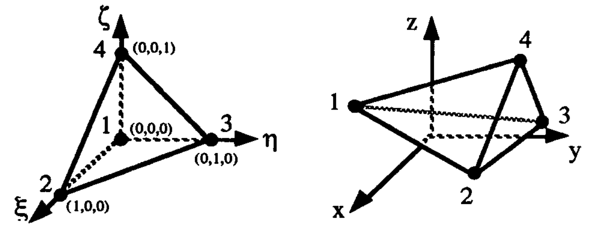


Figure 1. Tetrahedron geometry in natural (non-dimensional) and physical coordinate spaces.

The simplest interpolation function for tetrahedral cells is a linear one of the type in equation (4). It has four coefficients which, in this case, are expressed in terms of the nodal positions. The natural coordinates (ξ,η,ζ) are taken as per usual to vary from 0 to 1 in the non-dimensional cell (see Figure 1).

$$x(\xi,\eta,\zeta) = x_1 + (x_2-x_1)\xi + (x_3-x_1)\eta + (x_4-x_1)\zeta \quad (4)$$

This function does not have any non-linear terms like the trilinear function and can therefore be inverted analytically. This is done by expanding and rearranging (4) into the matrix form below. Note that x_p, y_p, and z_p are the coordinates of a given physical point and x_1, y_1, z_1,... etc. are the physical coordinates at the vertices of the tetrahedron.

$$\begin{pmatrix} x_p-x_1 \\ y_p-y_1 \\ z_p-z_1 \end{pmatrix} = \begin{pmatrix} x_2-x_1 & x_3-x_1 & x_4-x_1 \\ y_2-y_1 & y_3-y_1 & y_4-y_1 \\ z_2-z_1 & z_3-z_1 & z_4-z_1 \end{pmatrix} \begin{pmatrix} \xi \\ \eta \\ \zeta \end{pmatrix} \quad (5)$$

This system of equations can be solved by inverting the 3x3 matrix on the right and then premultiplying it with the vector on the left. The end result can be written as:

$$\begin{pmatrix} \xi \\ \eta \\ \zeta \end{pmatrix} = \frac{1}{V} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_p-x_1 \\ y_p-y_1 \\ z_p-z_1 \end{pmatrix} \quad (6)$$

where the constants in the 3x3 matrix are given by:

$$\begin{aligned}
a_{11} &= (z_4-z_1)(y_3-y_4) - (z_3-z_4)(y_4-y_1) \\
a_{21} &= (z_4-z_1)(y_1-y_2) - (z_1-z_2)(y_4-y_1) \\
a_{31} &= (z_2-z_3)(y_1-y_2) - (z_1-z_2)(y_2-y_3) \\
a_{12} &= (x_4-x_1)(z_3-z_4) - (x_3-x_4)(z_4-z_1) \\
a_{22} &= (x_4-x_1)(z_1-z_2) - (x_1-x_2)(z_4-z_1) \\
a_{32} &= (x_2-x_3)(z_1-z_2) - (x_1-x_2)(z_2-z_3) \\
a_{13} &= (y_4-y_1)(x_3-x_4) - (y_3-y_4)(x_4-x_1) \\
a_{23} &= (y_4-y_1)(x_1-x_2) - (y_1-y_2)(x_4-x_1) \\
a_{33} &= (y_2-y_3)(x_1-x_2) - (y_1-y_2)(x_2-x_3)
\end{aligned}$$

and the determinant, V (actually 6 times the volume of the tetrahedron), is given by:

$$\begin{aligned}
V = {} & (x_2-x_1)[\,(y_3-y_1)(z_4-z_1)-(z_3-z_1)(y_4-y_1)\,] \\
& + (x_3-x_1)[\,(y_1-y_2)(z_4-z_1)-(z_1-z_2)(y_4-y_1)\,] \\
& + (x_4-x_1)[\,(y_2-y_1)(z_3-z_1)-(z_2-z_1)(y_3-y_1)\,]
\end{aligned}$$

The natural coordinates (ξ,η,ζ) can be evaluated in 104 floating point operations by implementing the equations above. This figure can be halved by precomputing common terms before evaluating the matrix coefficients and determinant.

6. Tetrahedral decomposition

Equation (6) can be applied directly to unstructured grids containing tetrahedral cells. However, structured curvilinear grids containing hexahedral cells or hybrid grids containing mixed element types must be decomposed into tetrahedra in order to use it. Since the datasets for time-dependent flows are usually extremely large, it is impractical to do the decomposition as a preprocessing step. It must therefore be performed locally as particles enter cells.

A hexahedral cell can be divided into a minimum of five tetrahedra (Figure 2). This decomposition, however, is not unique because the diagonal edges alternate across a cell. Since the faces of a hexahedron are usually non-planar, it is important to ensure that adjoining cells have matching diagonals to prevent gaps. This is achieved by alternating between an odd and even decomposition as illustrated in Figure 2. In a structured grid, the correct configuration is selected by simply adding up the integer indices of a specific node (the node with the lowest indices was used in practice). Choosing the odd configuration when the sum is odd and the even configuration when the sum is even guarantees continuity between cells.

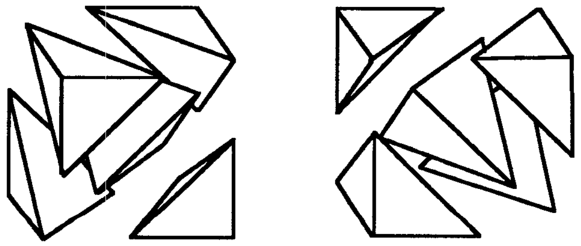


Figure 2. Sub-division alternates between two configurations to ensure continuity between cells.

7. Cell-search scheme

The previous sections discussed how to sub-divide hexahedral cells into tetrahedra and how to evaluate the natural coordinates of points within them. Those concepts are now combined with an efficient point location scheme to produce a cell-search technique which locates the tetrahedron that bounds the physical point (x_p,y_p,z_p).

Equation (6) allows the natural coordinates to be evaluated directly from the physical coordinates with relatively little effort. There are four conditions which must be valid for the point (x_p,y_p,z_p) to lie within the tetrahedron. They are:

$$\xi \geq 0 \ : \ \eta \geq 0 \ : \ \zeta \geq 0 \ : \ 1-\xi-\eta-\zeta \geq 0 \qquad (7)$$

If any one of these conditions is invalid then the point is outside the tetrahedron. In particle tracing algorithms, this happens when particles cross cell boundaries. The problem arises then of which tetrahedron to advance to next. The solution is quite simple since the natural coordinates tell you which direction to move. For example, if $\xi<0$, the particle would have crossed the $\xi=0$ face. Similarly, if $\eta<0$ or $\zeta<0$, the particle would have crossed the $\eta=0$ or $\zeta=0$ face respectively. If the fourth condition is violated, i.e. $(1-\xi-\eta-\zeta)<0$, then the particle would have crossed the diagonal face. The cell-search proceeds by advancing across the respective face into the adjoining tetrahedron.

Occasionally, two or more conditions in (7) may be violated if a particle crosses near the corner of a cell or if it traverses several cells at once. In such cases, the worst violator of the four conditions is used to predict which next tetrahedron to try next. Even if the bounding tetrahedron is not found in the immediate neighbour, by always moving in the direction of the worst violator it will converge upon the correct cell.

The cell search procedure described above should only be used if the cell being sought is nearby, that is, within a few cells of the previous one. This is usually the case during particle tracing since the majority of particles only cross one cell at a time. There are, however, two situations when the cell being sought is not likely to be nearby, these being: i) at the start of a particle trace and ii) after jumping between grids in multi-grid datasets. Under these circumstances the cell search should be preceded by another scheme in order to prevent weaving across a large grid one tetrahedron at a time. We use the 'boundary search' technique described by Buning [15].

8. Velocity interpolation in tetrahedra

The velocity components must be evaluated whenever a particle changes position or whenever the integration time step changes. They can be interpolated on the geometric cell in physical space using linear interpolation [14] or volume weighting [15], or on the unit cell in non-dimensional space using linear basis functions [16]. All three have been assessed in the present study and shown to be equivalent, i.e., they produce identical interpolation functions. However, one of them is much more efficient and ties in with point location algorithm already described.

8.1 Physical space linear interpolation

The physical space linear interpolation function can be written as:

$$u(x,y,z) = a_u + b_u x + c_u y + d_u z \qquad (8)$$

where u is one of the three velocity components and x, y, z the Cartesian coordinates. The constants a_u, b_u, c_u and d_u are evaluated by substituting the known values of u, x, y and z at each of the nodes into this expression. The resulting set of simultaneous equations can be written as:

$$\begin{pmatrix} a_u \\ b_u \\ c_u \\ d_u \end{pmatrix} = \begin{pmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{pmatrix}^{-1} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} \qquad (9)$$

where u_1, u_2, u_3, and u_4 are the velocity components at each of the four nodes. Note that the constants a, b, c, and d differ for each velocity component but the 4x4 matrix is the same for all because it only depends on the cell geometry. It turns out that nine of the coefficients in the inverted 4x4 matrix are identical to those in equation (6).

Using this approach, the velocity components can be evaluated in approximately 80 floating point operations assuming that the matrix coefficients are reused from the point location scheme.

8.2 Volume weighted interpolation

Volume weighting is a more popular interpolation technique and has been used in several particle tracing algorithms [10,15]. The interpolation strategy is to use the volumes of four sub-tetrahedra as weights applied to the velocity components at each node. These sub-tetrahedra are created by connecting the interpolation point, P say, to the four corners of the original tetrahedron. The ratio of the volume of each sub-tetrahedron to the volume of the whole tetrahedron gives the weight assigned to the vertex opposite the sub-tetrahedron. Using the notation in Figure 1, the volume weighted interpolation function at point P is given by:

$$u_P = u_1 \frac{V_{432P}}{V_{1234}} + u_2 \frac{V_{134P}}{V_{1234}} + u_3 \frac{V_{421P}}{V_{1234}} + u_4 \frac{V_{123P}}{V_{1234}} \qquad (10)$$

The volume of a generic tetrahedron, V_{ABCD}, is given by:

$$V_{ABCD} = \frac{1}{6}\left(\overrightarrow{AD} \cdot \left(\overrightarrow{AB} \times \overrightarrow{AC}\right)\right) \quad (11)$$

By expanding equation (10) using the volume definition in (11), it can be shown that this interpolation function is identical to the previous one. However, it takes 192 floating point operations to evaluate the four weights in equation (10) which makes it more expensive than the previous linear interpolation technique.

8.3 Linear basis function interpolation

The third and most popular technique for interpolating within a tetrahedron is the linear basis function approach. Using the same node numbering as before, the linear basis function can be written as:

$$u(\xi,\eta,\zeta) = u_1 + (u_2-u_1)\xi + (u_3-u_1)\eta + (u_4-u_1)\zeta \quad (12)$$

where ξ, η and ζ are the coordinates in the transformed tetrahedron (see Figure 1). By substituting the expressions for ξ, η, and ζ from equation (6) into (12), it can be shown that this interpolant is identical to that in (9). That is, both the linear physical and linear basis interpolation functions give the same interpolation result.

Since the natural coordinates are evaluated as part of the point location scheme, it is clearly more efficient to use the linear basis functions to evaluate the velocity components than either of the other two methods. The velocity components can be computed in just 27 floating point operations by using linear basis functions.

9. Numerical integration scheme

The numerical integration of equation (2) was performed with a standard 4th order Runge-Kutta scheme which comprises of the following stages:

$$\begin{aligned}
a &= \mathbf{v}(\,\mathbf{r}(t), t\,)\,\Delta t \\
b &= \mathbf{v}(\,\mathbf{r}(t)+\tfrac{a}{2}, t+\tfrac{\Delta t}{2}\,)\,\Delta t \\
c &= \mathbf{v}(\,\mathbf{r}(t)+\tfrac{b}{2}, t+\tfrac{\Delta t}{2}\,)\,\Delta t \\
d &= \mathbf{v}(\,\mathbf{r}(t)+c, t+\Delta t\,)\,\Delta t \\
\mathbf{r}(t+\Delta t) &= \mathbf{r}(t) + \frac{1}{6}(a + 2b + 2c + d) \quad (13)
\end{aligned}$$

where $\mathbf{r}$ is the particle position, $\mathbf{v}$ the velocity vector at that position, and Δt the time step which is set by the adaptive step size procedure described in the following section. A consequence of multi-stage schemes such as this one is that the velocities (and grid positions in a moving grid) must be frequently interpolated between the simulation time steps. In past [17] and present algorithms, these quantities are linearly interpolated. Darmofal and Haimes [13] have shown that linear interpolation in time reduces the temporal accuracy of the Runge-Kutta scheme from 4th to 2nd order.

The four stages of the Runge-Kutta scheme span three time values (t, t+Δt/2, and t+Δt) and therefore require new grid and velocity data at each one. Fortunately, these only need to be interpolated in the cell surrounding a particle. However, because a particle moves in time and space during the integration, it may lie in different cells at different stages of the integration, so point location is required after each stage.

There is an important difference in the way the integration is governed in unsteady flows compared with steady flows. In steady flows the integration is usually governed by the grid geometry, that is, particles are traced from one face to another across a cell. In unsteady flows the integration is governed by time and proceeds from one time step to the next. The point location and velocity interpolation procedures are called from the Runge-Kutta scheme as required. Step size adaption occurs after each complete integration step.

10. Curvature-based step size adaption

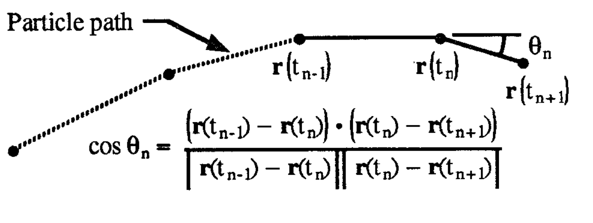
If the integration step size is fixed at a constant value along the entire particle path, or regulated to achieve a specified number of steps-per-cell, a particle may understeer around bends if the flow changes direction rapidly. This can be prevented by using an adaptive step size control scheme where the integration step size is changed according to an error tolerance.

The error tolerance can be computed using a standard numerical technique such as *step doubling* [18] whereby a particle is advanced forward from a given point using a step size Δt and then the process is repeated from the same point using two half steps of size Δt/2. The step size is reduced if the distance between the end-points is greater than a specified tolerance; a number usually deduced by trial and error.

We implemented the step doubling algorithm and tested it in several flows. The step size adaption worked well, particularly in the vicinity of critical points, although the performance was worse than a scheme which used 5 steps-per-cell. That was because the step doubling algorithm performed approximately twice the number of numerically expensive point locations.

A heuristic technique for adapting step size was suggested by Darmofal and Haimes [19]. They measured the angle between velocity vectors at successive points along a particle path to estimate the change in velocity direction. If the velocity direction changed too much (over 15 degrees) the step size was halved, whereas if it changed too little (less than 2 degrees) the step size was doubled.

We implemented this scheme and a very similar one which measured the angle between successive line segments on the path line (Figure 3). The latter scheme adapted the step size according to the path line curvature.



$$\cos\theta_n = \frac{\left(r(t_{n-1}) - r(t_n)\right) \cdot \left(r(t_n) - r(t_{n+1})\right)}{\left|r(t_{n-1}) - r(t_n)\right| \left|r(t_n) - r(t_{n+1})\right|}$$

Figure 3. Step adaption is based on the pathline curvature.

Both of these schemes worked well in practice and ran approximately three times faster than the step doubling algorithm. In both cases, the initial step size was estimated using:

$$\Delta t = \frac{1}{|\mathbf{u}|}\sqrt[3]{\frac{V}{6}} \qquad (14)$$

where $|\mathbf{u}|$ is the magnitude of the velocity at the current position and V is the determinant from equation (6). Following this initial estimate, the step size was halved if the angle θ_n was too acute (θ_n>15°), doubled if it was too obtuse (θ_n< 3°), or kept the same if it was in between these limits (3°≤ θ_n ≤ 15°).

Through experimentation in several datasets we found the curvature-based algorithm produced similar particle traces to the step-doubling scheme when adaption angles of 15 degrees (upper limit) and 3 degrees (lower limit) were used. The curvature-based algorithm produced particle traces of superior accuracy when the upper limit was lowered below 12 degrees.

11. Performance evaluation

The performance of the tetrahedral algorithm was compared with a conventional hexahedral algorithm [17] in two time-dependent datasets. The conventional algorithm used trilinear functions for spatial and velocity interpolation and an iterative Newton-Raphson method for point location. Both algorithms used 4th order Runge-Kutta integration schemes but used different step adaption techniques. The conventional algorithm used a semi-adaptive (5 steps-per-cell) scheme whereas the tetrahedral algorithm used the heuristic curvature-based method.

The missile dataset (Table 1) comprised of a single static grid with 500,000 grid points and 795 time-varying velocity fields. Each velocity field was approximately 4 MB (Megabytes) in size which yielded a total of 3.3 GB (Gigabytes) for the entire dataset. Sixteen particles were released at each simulation time step.

The V-22 tiltrotor dataset consisted of 26 grids (9 of which moved) containing 2.5 million grid points. We used 140 out of a total 1,400 simulation time steps, each one consisted of 100 MB of data. There were 120 particles released at each time step resulting in over 15,000 active particles being traced by the end of the simulation.

The execution times shown in Tables 1 and 2 have been divided up into the main algorithmic tasks and are given in both seconds and as a percentage of the total execution time. Computations were performed on a Convex 3420 with a striped disk system (16 heads).

Missile dataset (795 time steps)		
	Tetrahedral	Hexahedral
point location	503 (63.2%)	1159 (24.5%)
velocity interp.	45 (5.6%)	2058 (43.5%)
integration	73 (9.2%)	336 (7.1%)
step adaption	6 (0.7%)	445 (9.4%)
data manager	103 (13%)	662 (14.0%)
I/O & system	66 (8.3%)	71 (1.5%)
	796 (100%)	4731 (100%)

Table 1. Break-down of execution times for the new tetrahedral algorithm and a conventional hexahedral algorithm in a single-zone dataset. Times are given in seconds and as a percentage of the total execution time.

Table 1 shows that the tetrahedral algorithm ran about six times faster than the hexahedral algorithm in the missile dataset. Every aspect of algorithm is more efficient although the most significant gain comes from the velocity interpolation. This is because the point location and velocity interpolation are closely coupled in the new tetrahedral algorithm and the majority of the work, namely, computing the natural coordinates (ξ,η,ζ), is done during the point location.

In the V-22 dataset (Table 2), the speed advantage of the tetrahedral algorithm was obscured because of the addition of grid jumping; the process by which particles cross grids in a multi-grid dataset. Grid jumping is an infrequent process, it occurs on average once every 1,000 point locations, yet it takes the majority of the execution time in this case (57.4%). Results from the hexahedral algorithm show that the point location and velocity interpolation are the most expensive tasks, grid jumping is much less significant taking only 11% of the execution time. Clearly, the performance of the new tetrahedral algorithm could be significantly improved in multi-grid datasets with faster grid jumping code.

Future work will involve testing alternative grid jumping schemes such as the donor-receiver point method

[20] in which a pre-computed look-up table is used to predict where a particle 'lands' after a grid jump.

V-22 tiltrotor dataset (140 time steps)		
	Tetrahedral	Hexahedral
point location	928 (26.4%)	1250 (27.1%)
velocity interp.	77 (2.2%)	1139 (24.7%)
integration	105 (3.0%)	327 (7.1%)
step adaption	7 (0.2%)	277 (6.0%)
data manager	169 (4.8%)	890 (19.3%)
I/O & system	210 (6.0%)	221 (4.8%)
grid jumping	2018 (57.4%)	507 (11.0%)
	3514 (100%)	4611 (100%)

Table 2. Break-down of execution times for the tetrahedral and hexahedral algorithms in a multi-zone dataset.

Plates 1 and 2 show stills at time steps 550 and 654 from an animation of streak lines flowing around the missile. Streak lines generated using both the hexahedral (magenta dots) and tetrahedral (cyan lines) algorithms are shown in these pictures. As can be seen, there is virtually no difference between the particle paths generated by each algorithm.

Plates 3 and 4 show stills from an animation of the V-22 at time steps 56 and 140. The particles could not be rendered as streak lines because the flow was turbulent and had a great deal of recirculation which made it difficult to connect the points. Particles were released from a fixed location above the rotor blades which resulted in the distinct gaps as the blades cut through the descending sheet of particles. Considering the turbulent nature of this flow, there is very good agreement between particles computed with the tetrahedral and hexahedral algorithms.

Acknowledgements

Carl Hsieh of the Naval Surface Warefare Center provided the missile dataset and Robert Meakin of Overset Methods Inc. provided the V-22 dataset. We thank Michael Gerald-Yamasaki for the simple technique he suggested for choosing the appropriate cell sub-division, and all our other colleagues in the NAS Division for providing ideas and information. This work was supported by NASA under contract NAS 2-12961.

References

[1] A. Globus, A Software Model for Visualization of Large Unsteady 3-D CFD Results, AIAA 95-0115, AIAA 33rd Aerospace Sciences Meeting and Exhibit, Reno, 1995.

[2] D. Lane, Scientific Visualization of Large Scale Unsteady Fluid Flow, Scientific Visualization: Surveys, Methodologies and Techniques, ed. G.M. Nielson et al., IEEE Computer Society Press, 1996.

[3] S. Shirayama, Processing of Computed Vector Fields for Visualization, Journal of Computational Physics, Vol. 106, pp 30-41, 1993.

[4] R. Haimes, pv3: A Distributed System for Large-Scale Unsteady CFD Visualization, AIAA 94-0321, AIAA 32nd Aerospace Sciences Meeting and Exhibit, 1994.

[5] R. Lohner and J. Ambrosiano, A Vectorized Particle Tracer For Unstructured Grids, Journal of Computational Physics, Vol. 91, 1990, pp 22-31.

[6] D. Lane, Parallelizing a Particle Tracer for Flow Visualization, Proceedings from the Seventh SIAM Conference on Parallel Processing for Scientific Computing, 1995, pp 784-789.

[7] S. Bryson and C. Levit, The Virtual Windtunnel: An Environment for the Exploration of Three-dimensional Unsteady Flows, Proceedings of Visualization '91, IEEE Computer Society Press, 1991, pp 17-24.

[8] P. Eliasson, J. Oppelstrup, and A. Rizzi, Stream 3D: Computer Graphics Program for Streamline Visualization, Advances in Engineering Software, Vol. 11, No. 4, 1989, pp 162-168.

[9] P. Buning, Sources of Error in the Graphical Analysis of CFD Results, Journal of Scientific Computing, Vol. 3, Number 2, 1988, pp 149-164.

[10] A. Sadarjoen, T. van Walsum, A. Hin, and F. Post, Particle Tracing Algorithms for 3-D Curvilinear Grids, Proceedings of the 5th Eurographics Workshop on Visualization in Scientific Computing, Rostock, Germany, 1994.

[11] J. Hultquist, Interactive Numerical Flow Visualization Using Stream Surfaces, Ph.D. dissertation, University of North Carolina at Chapel Hill, May 1995.

[12] T. Pulliam, Efficient Solution Methods for Navier-Stokes Equations, Lecture notes from the von Karman Institute for Fluid Dynamics Lecture Series, Belgium, January 1986, pp 78.

[13] D. Darmofal and R. Haimes, An Analysis of 3-D Particle Path Integration Algorithms, Proceedings of the 1995 AIAA CFD Meeting, 1995.

[14] T. Chung, Finite Element Analysis in Fluid Dynamics, McGraw Hill, 1977, pp 60-93.

[15] P. Buning, Numerical Algorithms in CFD Post-Processing, in Compuer Graphics and Flow Visualization in Computational Fluid Dynamics, von Karman Institute for Fluid Dynamics Lecture Series, 1989-07, Sept 1989.

[16] G. Dhatt and G. Touzot, The Finite Element Method Displayed, J. Wiley, 1984, pp 110-113.

[17] D. Lane, Visualization of Time-Dependent Flow Fields, Proceedings of Visualization '93, IEEE Computer Society Press, 1993, pp 32-38.

[18] W. Press et al., Numerical Recipes, Cambridge University Press, 1986, pp 554-560.

[19] D. Darmofal and R. Haimes, Visualization of 3-D Vector Fields: Variations on a Stream, AIAA 92-0074, AIAA 30th Aerospace Sciences Meeting and Exhibit, Reno, Nevada, January 1992.

[20] J. Hultquist, Improving the Performance of Particle Tracing in Curvilinear Grids, AIAA 94-0324, AIAA 32nd Aerospace Sciences Meeting and Exhibit, Reno, Nevada, January 1994.

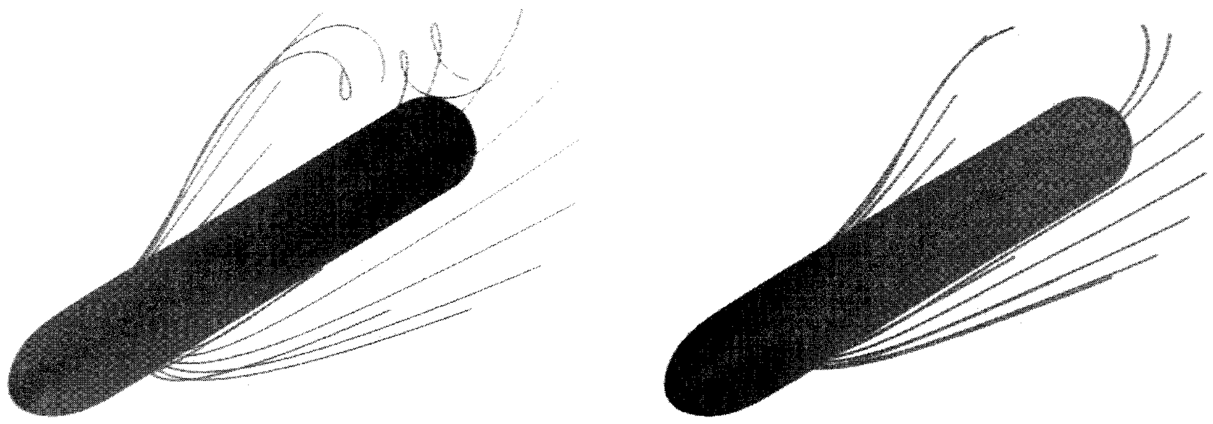


Plate 1. Streak lines from the missile dataset at time steps 550 and 654. Cyan lines depict those generated by the tetrahedral algorithm and magenta dots (unconnected for clarity) depict those by the hexahedral algorithm.

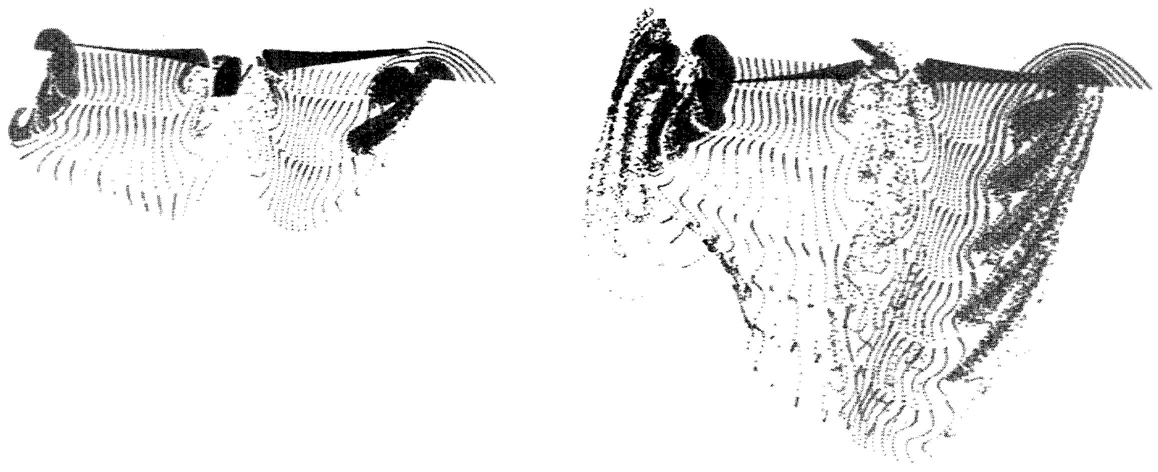


Plate 2. Time-dependent particle traces from the V-22 tiltrotor dataset at time steps 56 and 140. Particles generated by the tetrahedral and hexhedral algorithms are coloured cyan and magenta respectively.